ELSEVIER
Theoretical Computer Science 293 (2003) 219–236
Theoretical Computer Science
www.elsevier.com/locate/tcs

# Self-stabilization with path algebra

Bertrand Ducourthial[a,*], Sébastien Tixeuil[b]

[a] *Laboratoire HEUDIASYC, UMR CNRS 6599, Université de Technologie de Compiègne, 60205 Compiègne Cedex, France*
[b] *Laboratoire de Recherche en Informatique, UMR CNRS 8623, Université de Paris-Sud, 91405 Orsay Cedex, France*

**Abstract**

Self-stabilizing protocols can resist transient failures and guarantee system recovery in a finite time. We highlight the connexion between the formalism of self-stabilizing distributed systems and the formalism of generalized path algebra and asynchronous iterations with delay. We use the later to prove that a local condition on locally executed algorithm (being a strictly idempotent $r$-operator) ensures self-stabilization of the global system. As a result, a parametrized distributed algorithm applicable to any directed graph topology is proposed, and the function parameter of our algorithm is instantiated to produce distributed algorithms for both fundamental and high-level applications. Due to fault resilience properties of our algorithm, the resulting protocols are self-stabilizing at no additional cost. 

## 1. Introduction

*Self-stabilization*. Robustness is one of the most important requirements of modern distributed systems. Two approaches are possible to achieve fault tolerance: on the one hand, robust systems use redundancy to mask the effect of faults, on the other hand, *self-stabilizing* systems (see [11,25,31]) may temporarily exhibit an abnormal behavior, but must recover correct behavior within finite time. Self-stabilization copes with memory corruption, and with processors and links crash and restart (see [23]). This also means that the complicated task of initializing distributed systems is no longer needed, since self-stabilizing protocols regain correct behavior regardless of the initial state. The concern of several researchers is to demonstrate the applicability of the self-stabilization property to the current communication technology (for example, high-speed and mobile communication networks, see [9,16]).

* Corresponding author.
*E-mail addresses:* bertrand.ducourthial@hds.utc.fr (B. Ducourthial), tixeuil@lri.fr (S. Tixeuil).

0304-3975/03/$ - see front matter 
PII: S0304-3975(02)00238-4

*Related work*. Silent systems [12] are systems where the communication between the processors is fixed from some point of the execution. In our model, registers are used for communication between processors. Then a silent system has the property that the contents of the communication registers is not changed after some point in the execution. When the algorithm checks that a register needs to be changed before performing a write operation, all write operations may be eliminated when the silent system has reached a legitimate configuration. Silent systems are used to solve tasks such as leader election, spanning tree construction or single source shortest path algorithms. Note that several tasks fundamental to distributed systems are inherently non-silent. Such tasks include mutual exclusion or token passing, where the contents of communication registers have to change infinitely often in every possible execution of the system.

Historically, research in self-stabilization over general networks has mostly covered undirected networks where bidirectional communication is feasible (the Update protocol of [13], or the algorithms presented in [2,14]). Bidirectional communication is usually heavily used in bidirectional self-stabilizing systems to compare one node state with those of its neighbors and check for consistency. The self-stabilizing algorithms that are built upon the paradigm of local checking (see [5,6]) use this scheme. The lack of bidirectional communication was overcome in recent papers using several techniques. Strong connectivity (which is a weaker requirement than bidirectionality) was assumed to build a virtual well-known topology on which the self-stabilizing algorithm may be run (a tree in [1]). As many self-stabilizing algorithms exists for rings [11] or trees [3] in the literature, these constructions may be used to reuse existing algorithms in general networks.

The restriction of having either bidirectional communication media or strongly connected unidirectional networks are reasonable when the task to be solved is dynamic and the system is asynchronous: e.g. for traversal algorithms, a token has to be able to pass through every node infinitely often. However, there exists several silent tasks for which global communication is not required. For example, the single source shortest path task only requires that a directed path exists from a node called the *source* to any other node, but not the converse. Arora et al. [4] used the formalism of Iteration Systems to give sufficient conditions for convergence of systems solving related tasks. Silent tasks have been solved in a self-stabilizing way on directed graphs that are not strongly connected in [10], but the underlying network was assumed having no cycle (DAG). The absence of cycles permits to avoid cases where corrupted data moves forever in the system, preventing it from stabilizing.

*Our contribution*. In this paper, we concentrate on solving silent tasks in a self-stabilizing way on a truly general network, where no hypothesis are made about the strong connectivity or the presence of cycles. As in [4], our solution is by giving a condition on the distributed algorithm. However, in [4], the condition is given in terms of global system property, while our condition is independent of the task to be solved, and is only determined by the algebraic properties of the function computed locally by the algorithm.

The contribution of this paper is twofold. First we provide a way of modeling a class of silent self-stabilizing distributed algorithm through the formalism of max-plus

algebra, using a matrix representation of the system. Then we extend a result presented in [19] using this formalism. To this purpose, we provide a parametrized algorithm that can be instantiated with a local function. Our parameterized algorithm enables a set of silent tasks to be solved self-stabilizingly provided that these tasks can be expressed through local calculus operations called *r*-operators. The *r*-operators are general enough to permit applications such as shortest path calculus, depth-first-search tree construction, and ancestor list construction to be solved on arbitrary graphs while remaining self-stabilizing.

In addition, since our approach is condition based, there is no additional layer used to make an algorithm that satisfies this condition tolerant to transient failures. In fact, when no transient faults appear in the system, the performance suffers no overhead. Our system performs under the general fully distributed demon (see [28]).

*Outline of the paper*. The rest of the paper is organized as follows. In Section 2, we give some definitions pertinent to the protocols and proofs. The self-stabilizing parameterized protocol is presented in Section 3 along with the *r*-operators used for local computations and the matrix modeling based on path algebra and asynchronous iterations. The correctness reasoning for the parameterized protocol is given in Section 4. Applications to fundamental problems in distributed computing area are presented in Section 5. We discuss the extension of our ideas and make some concluding remarks in Section 6.

## 2. Self-stabilizing distributed systems

### 2.1. Underlying graph

A distributed system $\mathcal{S}$ is a collection of $N$ processors linked with communication media allowing them to exchange information. Such a system is modeled by a *directed graph* (also called *digraph*) $G(V,E)$, defined by a set of vertices $V$ and a set $E$ of edges $(v_1,v_2)$, which are ordered[1] pairs of vertices of $V$ $(v_1,v_2\in V)$. Each vertex $u$ in $V$ represents a processor $P_u$ of the system $\mathcal{S}$. Each edge $(u,v)$ in $E$, represents a communication link from $P_u$ to $P_v$ in $\mathcal{S}$. We give now some graph definitions.

The *in-degree* of a vertex $v$ of $G$ denoted by $\delta^-(v)$ is equal to the number of vertices $u$ such that the edge $(u,v)$ is in $E$. The incoming edges of each vertex $v$ of $G$ are numbered from 1 to $\delta^-(v)$.

A *directed path* from a vertex $v_0$ to a vertex $v_k$ in a digraph $G(V,E)$ is a list of consecutive edges of $E$, $(v_0,v_1),(v_1,v_2),\ldots,(v_{k-1},v_k)$. The length of this path is $k$. If each $v_i$ is unique in the path, the path is *elementary*. A *cycle* is a directed path where $v_0=v_k$. A digraph without any cycle is called a *directed acyclic graph* ($DAG$).

The *distance* between two vertices $u,v$ of a digraph $G$, denoted by $\mathrm{d}_G(u,v)$, is the minimum of the lengths of all directed paths from $u$ to $v$. The *diameter* of a digraph $G$, denoted by $\mathrm{Diam}(G)$, is the maximum of the distances between all couples of vertices

[1] $(v_1,v_2)\neq(v_2,v_1)$.

in $G$. The *strongly connected component* of a vertex $v$ in a digraph $G(V,E)$ is the set of all vertices $w$ of $V$ such that there exists a directed path from $v$ to $w$ and a directed path from $w$ to $v$. $G$ is *strongly connected* if it has exactly one strongly connected component.

The *direct descendants* of a vertex $v$ of a digraph $G(V,E)$ are all the vertices $w$ of $G$ such that the edge $(v,w)$ is in $E$. Their set is denoted by $\Gamma_G^{+1}(v)$. Similarly, the *direct ancestors* of a vertex $v$ of $G$ are all the vertices $u$ of $G$ such that the edge $(u,v)$ is in $E$. Their set is denoted by $\Gamma_G^{-1}(v)$. We denote by $\overline{\Gamma_G^{-1}(v)}$ the set $\Gamma_G^{-1}(v)\cup\{v\}$. The *ancestors* of $v$ are all the vertices $u$ such that there exists a path from $u$ to $v$. Their set is denoted by $\Gamma_G^{-}(v)$.

### 2.2. Communications and processors

A communication from processor $P_u$ to processor $P_v$ is only feasible if the vertex $u$ is a direct ancestor of the vertex $v$ in $G$ (i.e. $(u,v)$ is an edge of $G$). Such a communication is performed as follows. Processor $P_u$ writes the datum to be sent to $P_v$ into a dedicated shared register. Then $P_v$ is able to read the datum into this register and to use it. A processor may only write into its own shared register and can only read shared registers owned by its direct ancestor processors or itself.

Although we assume that any communication in the distributed system $\mathscr{S}$ is done through shared registers, Dolev [15] presented a transformation for simulating shared registers over unreliable bidirectional message passing communication channels in a self-stabilizing way.

In addition to those shared registers, processors may maintain local variables when executing their code. Such local variables are private to the processor and cannot be accessed by any of its neighbors.

A processor is a deterministic sequential machine that runs a single[2] process. The *state* of a processor is defined by the values of its local variables. The state of a link $(u,v)$ of $E$ is defined by the value of the associated shared register. A processor *action* (or step) consists of a `read` action, then an internal computation followed by a `write` action. Internal action of processors are not significant to its neighbors because they have no access to the variables that are manipulated by those actions. The `read` and `write` actions are the only way for two processors to communicate.

### 2.3. Configurations and executions

Classical definitions for configurations and executions of distributed systems can be found in [27]. A *configuration* of a distributed system $\mathscr{S}$ is an instance of the states of its processors and links. The set of configurations of $\mathscr{S}$ is denoted as $\mathscr{C}$. Processor actions change the global system configuration. An *execution* $e$ (also called a

[2] The case of a processor scheduling several communicating processes is handled by considering those as *virtual* processors, each running a single process.

*computation*) is a sequence of configurations $c_1, c_2, \ldots$ such that for $i = 1, 2, \ldots$, the configuration $c_{i+1}$ is reached from $c_i$ by a single step of at least one processor. Configuration $c_1$ is the *initial configuration* of execution $e$.

The set of executions in the distributed system $\mathcal{S}$ starting with a particular initial configuration $c_1 \in \mathcal{C}$ is denoted by $\mathcal{E}_{c_1}$. Every execution $e \in \mathcal{E}_{c_1}$ is of the form $c_1, c_2, \ldots$ . The set of executions in system $\mathcal{S}$ whose initial configurations are all elements of $\mathcal{C}_1 \subset \mathcal{C}$ is denoted as $\mathcal{E}_{\mathcal{C}_1}$. The set $\mathcal{E} = \mathcal{E}_{\mathcal{C}}$ contains all possible executions of system $\mathcal{S}$. All executions considered in this paper are assumed to be *maximal* meaning that the sequence is either infinite, or it is finite and no action is enabled in the final configuration. An algorithm is *silent* if for each possible execution, either of the two following conditions is verified: (i) the execution is finite or (ii) the execution is infinite and there exists a configuration $c_t$ such that any subsequent execution (in $\mathcal{E}_{c_t}$) contains only $c_t$ configurations.

To model the non-deterministic behavior of a distributed system, we assume processor activity is managed by a global *scheduler*. To ensure correctness of the system, we regard the scheduler as an adversary. Each processor that is chosen by the adversary executes exactly one atomic step. The adversary may be more or less powerful depending on (i) the freedom it has in choosing the activated processors and (ii) the grain of the atomicity. More freedom and finer atomicity grain gives the adversary more power. We refer to the most common types of adversaries used in the literature and more specifically in [14,22,28]:

(1) The *synchronous demon* activates simultaneously all of the system's processors. Then, all processors read their input registers, perform local computations and write their output register.
(2) The *distributed demon* can activate simultaneously any subset of the system's processors. When such a subset is activated, all processors in the subset read their input registers, perform local computations and write their output register.
(3) The *fully distributed demon* can activate simultaneously any subset of the system's processor. When such a subset is activated, all processors in the subset read their input registers, perform local computations, but may delay writing their output register after the following demon activation occurred. However, a processor may not be chosen again by the demon until it has written its output registers.
(4) The *read/write demon* activates a single processor at a time. When a processor is activated, it may either read exactly one input register or (not both) write its output register. Contrary to the preceding demons that use composite atomicity (any processor read *all* its input registers atomically, and write *all* its output registers atomically), the Read/Write demon uses register atomicity.

The strongest adversary is the Read/Write demon, while the weakest adversary is the Synchronous demon. As an immediate corollary, if a distributed system works correctly under the scheduling of the strongest demon, it will also perform correctly under a weaker adversary.

**Hypothesis 1.** *We assume an intermediate adversary, the fully distributed demon.*

### 2.4. Self-stabilization

A *specification* is a predicate on executions that are admissible for a distributed system. A system *matches its specification* if all its possible executions match the specification. If we consider only *static* problems (i.e., problems whose solutions consist of computing some global result), the specification can be given in terms of a set of configurations. Every execution matching the specification would be a sequence of such configurations. The set of configurations that matches the specification of static problems is called the set of *legitimate* configurations (denoted as $\mathscr{L}$), while the remainder $\mathscr{C}\setminus\mathscr{L}$ denotes the set of *illegitimate* configurations. Self-stabilization is defined through the concept of closed attractor.

**Definition 2** (Closed Attractor). Let $\mathscr{C}_a$ and $\mathscr{C}_b$ be subsets of $\mathscr{C}$. $\mathscr{C}_a$ is an attractor for $\mathscr{C}_b$ if and only if for any initial configuration $c_1$ in $\mathscr{C}_b$, for any execution $e$ in $\mathscr{E}_{c_1}$, $(e=c_1,c_2,\ldots)$, there exists $i\geqslant 1$ such that for any $j\geqslant i$, $c_j\in\mathscr{C}_a$.

In the usual (i.e. non-stabilizing) distributed systems, possible executions can be restricted by allowing the system to start only from some well-defined *initial* configurations. On the other hand, in stabilizing systems, problems cannot be solved using this convenience, since all possible system configurations are admissible initial configurations.

**Definition 3** (Self-stabilization). A system $\mathscr{S}$ is called self-stabilizing if and only if there exists a non-empty subset $\mathscr{L}\subset\mathscr{C}$ of legitimate configurations such that $\mathscr{L}$ is a closed attractor for $\mathscr{C}$.

## 3. Self-stabilizing global computations with path algebra

The purpose of this section is twofold. First we recall previous results concerning relations between distributed systems and operator-based algorithms on the one hand, and between function-weighted graphs and path algebra for the other hand. Then we extend previous results to model silent distributed systems using the matrix representation used in asynchronous iterations.

### 3.1. Silent distributed systems as operator-based algorithms

In this section, we describe distributed systems that perform a global calculus using a parametric algorithm that simply gets input data from its incoming neighbors, computes a local function $\mathscr{F}_v$ and finally makes the result of this function available to its outgoing neighbors. Since silent distributed algorithms have their communication fixed from some point in each execution, we can consider this point as the global result computed by the algorithm.

The program for each protocol consists of a *rule* of the form: $\langle guard\rangle\rightarrow statement$. A *guard* is a boolean expression over the local variables of a processor and the

communication registers of its immediate ancestors. A *statement* is allowed to update the communication register of the processor only. Any rule whose guard is *true* is *enabled.*

**Hypothesis 4.** *We assume a* fair *adversary*, *i.e. in any infinite execution*, *if a processor has a rule that is enabled infinitely often*, *then this processor is chosen by the adversary infinitely often.*

Each processor $P_v$ has two local constants stored in Read Only Memory: the *initial datum*, $\mathsf{ROM}[v]$, and the set of its direct ancestors $\Gamma_G^{-1}(v)$. To store the result of the local computation, a single register is used at $P_v$: $\mathsf{RES}[v]$, the *outgoing variable*. Such a register is used for the communications between $P_v$ and all its direct descendants $P_w$ (*one-to-many communication scheme*). Each processor $P_v$ also has access to the communication register $\mathsf{RES}[u]$ of any of its direct ancestors $u \in \Gamma_G^{-1}(v)$, the *incoming variables*. In addition to the above, the protocol maintains on each processor $P_v$ a function $\mathscr{F}_v$ which is defined as

$$\begin{array}{rll} \mathscr{F}_v : & \mathtt{A}^{\delta^-(v)+1} & \to \mathtt{A} \\ & \mathsf{ROM}[v], \mathsf{RES}[u_1], \ldots, \mathsf{RES}[u_{\delta^-(v)}] & \mapsto \mathscr{F}(\mathsf{ROM}[v], \mathsf{RES}[u_1], \ldots, \mathsf{RES}[u_{\delta^-(v)}]) \end{array}$$

where nodes $u_1$ through $u_{\delta^-(v)}$ are the direct ancestors of $v$. Then each processors $v$ runs as follows: $v$ performs a local computation using function $\mathscr{F}_v$, its initial datum $\mathsf{ROM}[v]$ and the incoming data $\mathsf{RES}[u_1], \ldots, \mathsf{RES}[u_{\delta^-(v)}]$, and then stores the result into its outgoing variable $\mathsf{RES}[u]$. This guarded rule is parametrized at each node $v$ by $\mathscr{F}_v$ and is shown in Algorithm 1.

**Algorithm 1** (The $\mathscr{PA}$ algorithm). *Algorithm $\mathscr{PA}$ at node $v$ consists in one rule $R$ parametrized by $\mathscr{F}_v$*

$R|\mathscr{F}_v : \langle \mathsf{true} \rangle$
$\to \quad \mathsf{RES}[v] \leftarrow \mathscr{F}_v(\mathsf{ROM}[v], \mathsf{RES}_{u \in \Gamma_G^{-1}(v)}[u])$

*where* $\mathsf{RES}_{u \in \Gamma_G^{-1}(v)}[u]$ *stands for the sequence* $\mathsf{RES}[u_1], \ldots, \mathsf{RES}[u_{\delta^-(v)}]$ *with* $u_1$ *through* $u_{\delta^-(v)}$ *being the direct ancestors of* $v$.

**Note.** Having each node $v$ disposing of the $\Gamma_G^{-1}(v)$ is only convenient when writing the algorithm. In an actual implementation, this set can be efficiently replaced by the list of $v$ in-port hardware addresses.

### 3.2. r-operators

Our distributed algorithm is parametrized at each node $v$ in system $\mathscr{S}$ with function $\mathscr{F}_v$. The $\mathscr{F}_v$ function computes a result from $v$ direct ancestors' values. In Section 5, we only define functions $\mathscr{F}_v$ for each $v \in \mathscr{S}$ to describe the whole system. In the following, we investigate sufficient conditions on the $\mathscr{F}_v$ functions so that the system is self-stabilizing for a given specification.

*3.2.1. Infimum functions*

In [26], Tel proves that infimum computations terminate when the $\mathscr{F}_v$ function is an infimum over the set of inputs. An *infimum* (hereby called an *s-operator*) $\oplus$ over the set $\mathbb{S}$ is an associative, commutative and idempotent binary operator. Such an operator defines a partial order relation $\preccurlyeq_\oplus$ over the set $\mathbb{S}$ by: $x \preccurlyeq_\oplus y$ if and only if $x \oplus y = x$. Moreover, [26] assumes that there exists a greatest element on $\mathbb{S}$, denoted by $\top$, and verifying $x \preccurlyeq_\oplus \top$ for every $x \in \mathbb{S}$. If necessary, this element can be added to $\mathbb{S}$.

Hence, the $(\mathbb{S}, \oplus)$ structure is an *abelian idempotent semigroup*[3]. Using $\oplus$ as $\mathscr{F}_v$ in our parametrized algorithm yields a silent distributed system, yet [19] proved that the resulting protocol is not self-stabilizing.

*3.2.2. Binary r-operators*

Starting from Tel results, Ducourthial [17] introduced a distorted algebra—the *r*-algebra—by generalizing properties of the abelian idempotent semigroup with a mapping *r*. An *r-operator* is a dissymmetric *s*-operator, that we usually denote by $\triangleleft$:

**Definition 5** (*r*-operator). The operator $\triangleleft$ is an *r*-operator on $\mathbb{S}$ if there exists a bijective mapping $\mathsf{r}$ from $\mathbb{S}$ to $\mathbb{S}$ such that $\triangleleft$ verifies the following properties: (a) *r*-associativity: $(x \triangleleft y) \triangleleft \mathsf{r}(z) = x \triangleleft (y \triangleleft z)$; (b) *r*-commutativity: $\mathsf{r}(x) \triangleleft y = \mathsf{r}(y) \triangleleft x$; (c) *r*-idempotency: $\mathsf{r}(x) \triangleleft x = \mathsf{r}(x)$.

For example, the operator $\mathsf{minc}(x, y) = \mathsf{min}(x, y+1)$ is an idempotent *r*-operator on $\mathbb{Z} \cup \{+\infty\}$. The mapping $\mathsf{r}$ (which is $x \mapsto x+1$ for $\mathsf{minc}$) is called *r*-mapping of the *r*-operator. This mapping is unique and when it is equal to the identity ($x \mapsto x$), the corresponding *r*-operator is an $\mathsf{s}$-operator. The *r*-operators have many applications in parallel and distributed computing (see [17,18] for further details). We recall some of their algebraic properties. For any *r*-operator, there exists an *s*-operator $\oplus$ such that for any $x$ and $y$ in $\mathbb{S}$, $x \oplus y = x \triangleleft \mathsf{r}^{-1}(y)$. The identity element $\top$ of $\oplus$ is the right identity element of $\triangleleft$. Moreover, the *r*-mapping of any *r*-operator is an isomorphism of $(\mathbb{S}, \triangleleft)$ and $(\mathbb{S}, \oplus)$. There are as many *r*-operators on $\mathbb{S}$ as couples of *s*-operators $\oplus$ and isomorphisms $\mathsf{r}$ of $(\mathbb{S}, \oplus)$. When the *r*-operator $\triangleleft$ verifies $x \triangleleft x = x$ for any $x$ in $\mathbb{S}$, it is *idempotent*. Constructing an idempotent *r*-operator $\triangleleft$ from an *s*-operator $\oplus$ and an isomorphism $\mathsf{r}$ of $(\mathbb{S}, \oplus)$ is done by assuming that $\oplus$ and $\mathsf{r}$ verify, for any $x$ in $\mathbb{S}$, $x \preccurlyeq_\oplus \mathsf{r}(x)$. When for each $x$ in $\mathbb{S}$, $x \prec_\oplus \mathsf{r}(x)$ (i.e. $x \preccurlyeq_\oplus y$ and $x \neq y$), the *r*-operator $\triangleleft$ is *strictly idempotent*. For some proofs, we suppose that the *r*-operator $\triangleleft$ verifies: $\forall y, z \in \mathbb{S}$, $(\forall x \in \mathbb{S},\ x \triangleleft y = x \triangleleft z) \Leftrightarrow (y = z)$. In practice, this hypothesis is verified by many operators.

*3.2.3. n-ary r-operators*

Binary *r*-operators can be extended to accept an arbitrary number of arguments. A mapping $\triangleleft$ from $\mathbb{S}^n$ into $\mathbb{S}$ is an *n-ary r-operator* if there exists an *s*-operator $\oplus$

[3] The prefix *semi* means that the structure cannot be completed to obtain a group, since the law $\oplus$ is idempotent.

on $\mathbb{S}$ and $n-1$ isomorphisms $\mathsf{r}_1,\ldots,\mathsf{r}_{n-1}$ of $(\mathbb{S},\oplus)$ such that

$$\triangleleft(x_0,\ldots,x_{n-1}) = x_0 \oplus \mathsf{r}_1(x_1) \oplus \cdots \oplus \mathsf{r}_{n-1}(x_{n-1})$$

for any $x_0,\ldots,x_{n-1}$ in $\mathbb{S}$. In other words, an $n$-ary $r$-operator consists in $n-1$ binary $r$-operators based on the same $s$-operator. If all of these binary $r$-operators are (strictly) idempotent, the resulting $n$-ary $r$-operator is (strictly) idempotent.

**Hypothesis 6** ($r$-operator). *An $n$-ary $r$-operator $\triangleleft$ is defined on $\mathbb{S}$ from an $s$-operator $\oplus$ on $\mathbb{S}$, and $n-1$ endomorphisms $\mathsf{r}_1,\ldots,\mathsf{r}_{n-1}$ of $(\mathbb{S},\oplus)$ as follows: for all $x_0,\ldots,x_{n-1}\in\mathbb{S}$, $\triangleleft(x_0,\ldots,x_{n-1})=x_0\oplus\mathsf{r}_1(x_1)\oplus\cdots\oplus\mathsf{r}_{n-1}(x_{n-1})$.*

In addition, we suppose that the following properties on the $r$-mappings hold:

**Hypothesis 7** (Strict idempotency). *For any $r$-mapping $\mathsf{r}_i$ used in the distributed system $\mathcal{S}$ and for any $x\in\mathbb{S}$, $x\prec_{\oplus}\mathsf{r}_i(x)$.*

**Hypothesis 8** ($\mathsf{r}_\perp$). *Let $\mathsf{r}_\perp$ be the endomorphism built using all of the $r$-mappings $\mathsf{r}_1,\ldots,\mathsf{r}_{|E|}$ used in the distributed system $\mathcal{S}$ as follows: $\forall x\in\mathbb{S}$, $\mathsf{r}_\perp(x)=\bigoplus_{i\in\{1,\ldots,|E|\}}\mathsf{r}_i(x)$. Then $\lim_{k\to+\infty}\mathsf{r}_\perp^k(x)=\mathsf{e}_\oplus$.*

*3.2.4. Self-stabilization with r-operators*

When our distributed algorithm is instantiated with an $r$-operator $\triangleleft$ as function $\mathcal{F}_v$, each node $v\in\mathcal{S}$ performs:

$$\mathsf{RES}[v] = \mathsf{ROM}[v] \oplus \mathsf{r}(\mathsf{RES}[u_1]) \oplus \cdots \oplus \mathsf{r}(\mathsf{RES}[u_{\delta^-(v)}]) \qquad \textit{binary } r\textit{-operator},$$
$$\mathsf{RES}[v] = \mathsf{ROM}[v] \oplus \mathsf{r}_1(\mathsf{RES}[u_1]) \oplus \cdots \oplus \mathsf{r}_{\delta^-(v)}(\mathsf{RES}[u_{\delta^-(v)}]) \quad \textit{n-ary } r\textit{-operator},$$

where $u_1,\ldots,u_{\delta^-(v)}$ denote node $v$ direct ancestors, $\mathsf{r}_1,\ldots,\mathsf{r}_{\delta^-(v)}$ denote the corresponding $r$-mappings, and $\oplus$ denotes the $s$-operator $\triangleleft$ is based upon.

Ducourthial [18] showed that when the r-operator is idempotent, the algorithm is silent, and Ducourthial and Tixeuil [19] proved that when it is strictly idempotent and $\oplus$ induces a total order on $\mathbb{S}$, it is also self-stabilizing for the following specifications:

$$\mathsf{RES}[v] = \bigoplus\{\mathsf{r}^{\mathrm{d}_G(u,v)}(\mathsf{RES}[u]), u\in\overline{\Gamma_G^-(v)}\} \qquad \textit{binary},$$
$$\mathsf{RES}[v] = \bigoplus\{\mathsf{r}_P(\mathsf{RES}[u]), u\in\overline{\Gamma_G^-(v)},\ P \text{ elementary path from } u \text{ to } v\} \quad \textit{n-ary},$$

where $\mathsf{r}_P$ is the composition of the $r$-mappings corresponding to the edges of the path $P$. The proof of stabilization in [19] has been established for a read/write demon, assuming the $s$-operator $\oplus$ defined a total order relation. In this paper, we extend this result and prove—using path algebra—that even if the $s$-operator $\triangleleft$ is based upon defines a partial order relation on $\mathbb{S}$, our parametric algorithm instantiated with $\triangleleft$ is self-stabilizing. This extension leads to new applications, that we present in Section 5. However, we restrict executions by assuming a fully distributed demon.

### 3.3. Path algebra

*Semi-algebra.* In [7,20,21], definitions related to max-plus algebra can be found. A *semiring* $(\mathbb{S},\oplus,\otimes)$ is defined by the four following conditions: (a) $(\mathbb{S},\oplus)$ is an abelian semigroup whose identity element is $e_\oplus$, (b) $\otimes$ is associative and admits an identity element $e_\otimes$, (c) $\otimes$ is distributive over $\oplus$ and (d) $e_\oplus$ is absorbing for $\otimes$. The structure is idempotent when $\oplus$ is idempotent.

A *semimodule* $(\mathbb{M},\oplus,\cdot)$ over the semiring $(\mathbb{S},\oplus,\otimes)$ is a set of matrices $\mathbf{A}$ such that the following conditions hold: (a) $(\mathbb{M},\oplus)$ is an abelian semigroup which $\oplus$ is defined by $(\mathbf{A}\oplus\mathbf{B})[i][j]=\mathbf{A}[i][j]\oplus\mathbf{B}[i][j]$ and which identity element is denoted as $\mathbf{E}_\oplus$, (b) $\cdot$ is an external composition law defined by $(\lambda\cdot\mathbf{A})[i][j]=\lambda\cdot\mathbf{A}[i][j]$ with $\lambda\in\mathbb{S}$ and $\mathbf{A}\in\mathbb{M}$, (c) $\oplus$ and $\cdot$ are distributive and $\cdot$ verifies $\alpha(\beta\mathbf{A})=\beta(\alpha\mathbf{A})$ for all $\alpha$ and $\beta$ in $\mathbb{S}$ and (d) $e_\oplus\mathbf{A}=\mathbf{E}_\oplus$ and $e_\otimes\mathbf{A}=\mathbf{A}$.

Let $\otimes$ define a matrix multiplication: $(\mathbf{A}\otimes\mathbf{B})[i][j]=\bigoplus_{k=1}^{k=N}\mathbf{A}[i][k]\otimes\mathbf{B}[k][j]$. When endowed with this third law $\otimes$ such that $(\mathbb{M},\oplus,\otimes)$ is a semiring, $(\mathbb{M},\oplus,\otimes,\cdot)$ is a *semi-algebra*. The identity element of $\otimes$ is denoted by $\mathbf{E}_\otimes$. When $(\mathbb{S},\oplus)$ is idempotent, the semi-algebra is also idempotent.

*Graph interpretation.* There exists a duality between matrix operations and graph theory. The part of max-plus algebra that studies graph theory is known in the literature as *path algebra*.

The *precedence graph* $G[\mathbf{A}]$ associated to an $N\times N$ matrix $\mathbf{A}$ of the semi-algebra $(\mathbb{M},\oplus,\otimes,\cdot)$ is a directed graph with $N$ vertices numbered from 1 to $N$, such that there exists an arc from $j$ to $i$ if and only if $\mathbf{A}[i][j]\neq e_\oplus$. When different from $e_\oplus$, $\mathbf{A}[i][j]$ is called *weight* of the edge $(j,i)$. The weight $w(P)$ of a directed path $P=(v_0,v_1)\ldots(v_{k-1},v_k)$ in $G[\mathbf{A}]$ is equal to the product by $\otimes$ of the weights of the edges of $P$: $w(P)=\mathbf{A}[v_1][v_0]\otimes\cdots\otimes\mathbf{A}[v_k][v_{k-1}]$. We use the following standard notations: $\mathbf{A}^{(k)}=\mathbf{E}_\otimes\oplus\mathbf{A}\oplus\cdots\oplus\mathbf{A}^k$ and $\mathbf{A}^*=\lim_{k\to+\infty}\mathbf{A}^{(k)}$. A circuit $C$ is *absorbing* if its weight $w(C)$ verifies $e_\otimes\oplus w(C)\neq e_\otimes$.

In [20], Gondran showed that if $\mathbf{A}$ is an $N\times N$ matrix in a semi-algebra, the following properties hold:

(1) $(\mathbf{A}^k)[i][j]$ is the sum (in sense of $\oplus$) of the weights of the paths from $j$ to $i$ having exactly $k$ edges.
(2) $(\mathbf{A}^{(k)})[i][j]$ is the sum of the weights of the paths from $j$ to $i$ having at most $k$ edges.
(3) if $G[\mathbf{A}]$ has no absorbing circuit, then there exists an integer $p<N$ such that $\mathbf{A}^*=\mathbf{A}^{(p)}$.

*Generalized path algebra.* In [21,24], an extension was provided, allowing to label edges with a function instead of a scalar. Let $(\mathbb{S},\oplus)$ be an abelian idempotent semigroup, with $e_\oplus$ as its neutral element, and let $\mathbb{H}$ be the set of the endomorphisms over $(\mathbb{S},\oplus)$. Let the $\oplus$ law of $\mathbb{S}$ be extended to $\mathbb{H}$ by $(\mathsf{h}_1\oplus\mathsf{h}_2)(x)=\mathsf{h}_1(x)\oplus\mathsf{h}_2(x)$ and let $\mathsf{e}_\oplus\in\mathbb{H}$ (defined by $\mathsf{e}_\oplus(x)=e_\oplus$ for all $x\in\mathbb{S}$) be its neutral element. Let $\otimes$ be the composition law of two morphisms of $\mathbb{H}:(\mathsf{h}_1\otimes\mathsf{h}_2)(x)=\mathsf{h}_2(\mathsf{h}_1(x))$ and let $\mathsf{e}_\otimes:x\mapsto x$ be its neutral element. Then $(\mathbb{H},\oplus,\otimes)$ is an idempotent semiring.

In particular, given these notations, if for each edge $(i,j)$ in the graph $G[\mathbf{A}]$, its label $\mathsf{h}_{ij} \in \mathbb{H}$ verifies $\mathsf{e}_\otimes \oplus \mathsf{h}_{ij} = \mathsf{e}_\otimes$, the graph has no absorbing circuit, and $\mathbf{A}^*$ exists.

### 3.4. Silent distributed systems as asynchronous iterations

#### 3.4.1. System configurations as vectors

The distributed systems we consider have processors communicating through shared registers. Since we consider executions under the synchronous, distributed and fully distributed demons, which use composite atomicity, processors read all their incoming registers in a single atomic step. Since processors hold a single output register, a given configuration of the system is modeled by a vector of register values (one entry per processor). During an execution, we have a sequence of such vectors, where $\mathtt{X}_n$ denotes the $n$th vector describing the system configuration. The successive evolutions of the system during each executions are properly described through the corresponding evolutions of the configuration vector.

We now review the synchronous, distributed and fully distributed demons, and their relationship with asynchronous iterations.

#### 3.4.2. The synchronous demon

As detailed in Section 3.2, when parametrized by an $n$-ary $r$-operator $\triangleleft$ based on an $s$-operator $\oplus$, our distributed algorithm leads to the following local computations:

$$\mathsf{RES}[v] = \mathsf{ROM}[v] \oplus \mathsf{r}_1(\mathsf{RES}[u_1]) \oplus \cdots \oplus \mathsf{r}_{\delta^-(v)}(\mathsf{RES}[u_{\delta^-(v)}]), \tag{1}$$

Under the synchronous demon, all nodes perform this local computation simultaneously. Thus, the system computations can be written using a matrix notation. Let $(\mathbb{S}, \oplus)$ be the idempotent abelian semigroup corresponding to the $s$-operator associated to the $n$-ary $r$-operator $\triangleleft$. Let $(\mathbb{H}, \oplus, \otimes)$ be the idempotent semiring of the endomorphisms of $\mathbb{S}$, let $\mathsf{e}_\oplus$ and $\mathsf{e}_\otimes$ be its zero (neutral element for $\oplus$) and unity (neutral element for $\otimes$), respectively. Let $(\mathbb{M}, \oplus, \otimes, \cdot)$ be the idempotent semi-algebra over $(\mathbb{H}, \oplus, \otimes)$ of the $N \times N$ matrices (see Section 3.3). Now consider the $N \times N$ precedence matrix $\mathbf{A} \in \mathbb{M}$, associated to system $\mathscr{S}$ (composed of $N$ nodes): $\mathbf{A}[i][j] = \mathsf{e}_\oplus$ if the edge $(i,j)$ does not exist, and $\mathbf{A}[j][i] = \mathsf{r}_{ij}$ if the edge $(i,j)$ exists and $\mathsf{r}_{ij}$ is its $r$-mapping.

Let $\mathtt{X}_n$ be the vector composed of the values stored in the outgoing registers of each of the $N$ nodes of system $\mathscr{S}$ at step $n$. Let $\mathtt{B}$ be the vector composed of the values stored in the ROMs of those nodes in the same order. Then, we have

$$\mathtt{X}_{n+1} = \mathbf{A} \otimes \mathtt{X}_n \oplus \mathtt{B}. \tag{2}$$

In the following, $\mathsf{F}$ denotes the vector operator: $\mathtt{X} \mapsto \mathsf{F}(\mathtt{X}) = \mathbf{A} \otimes \mathtt{X} \oplus \mathtt{B}$.

#### 3.4.3. The distributed demon

The distributed demon does not necessarily activate all processors at the same time. Thus, Eq. (2) does not hold in this case. Still it is possible to consider global

computations steps, but we must take into account that all processors may not participate in building a new value at each step. Let $\mathscr{J}_n$ be the set of processors activated at step $n$ by the distributed demon. From Eq. (2), we obtain

$$\mathtt{X}_{n+1}[i] = \begin{cases} \mathtt{X}_n[i] & \text{if} \quad i \notin \mathscr{J}_n, \\ (\mathtt{F}(\mathtt{X}_n))[i] = (\mathbf{A} \otimes \mathtt{X}_n \oplus \mathtt{B})[i] & \text{if} \quad i \in \mathscr{J}_n. \end{cases} \tag{3}$$

Informally, Eq. (3) reads as only selected processors (those of $\mathscr{J}_n$) compute a new value using the last produced result of their direct ancestors. Such equations are known in the literature as *asynchronous iterations* (see [8,30]).

### 3.4.4. The fully distributed demon

Under control of the fully distributed demon, activated processors do not necessarily write their output registers within the same round. Thus, Eq. (3) does not hold in this context. Hence, a processor may compute its output value using its last read input values, while those input values have changed in between. This enforces the asynchronism between processors because they do not have to perform similar operations (computing their local algorithm) within the same time (as it is the case under the synchronous or distributed demon).

We introduce a delay in Eq. (3) to model this time lap between `read` and `write` actions. At step $n$, supposing $i \in \mathscr{J}_n$, processor $P_i$ uses the data of its direct ancestor $P_j$ produced at step $D_n[j]$ (instead of $n-1$). Delaying leads to the following *asynchronous iteration with delay* (see [30]):

$$\mathtt{X}_{n+1}[i] = \begin{cases} \mathtt{X}_n[i] & \text{if} \quad i \notin \mathscr{J}_n, \\ (\mathtt{F}((\mathtt{X}_{D_n[1]}[1], \ldots, \mathtt{X}_{D_n[N]}[N])^{\mathrm{t}}))[i] & \\ \quad = (\mathbf{A} \otimes (\mathtt{X}_{D_n[1]}[1], \ldots, \mathtt{X}_{D_n[N]}[N])^{\mathrm{t}} \oplus \mathtt{B})[i] & \text{if} \quad i \in \mathscr{J}_n. \end{cases} \tag{4}$$

### 3.4.5. Conditions for convergence of asynchronous iterations

Asynchronous iterations have been extensively studied for optimization purpose on parallel computers (see [8,29,30]). Under particular conditions, asynchronous iterations (Eqs. (3) and (4)) converge to the same result as *synchronous iterations* (Eq. (2)), while reducing data dependency.

In [30], Üresin and Dubois give several sufficient conditions ensuring the convergence of asynchronous iterations. They also point out that their work can be applied to path algebra. Since the vector operator $\mathtt{F}$ is defined on a Cartesian product of (possibly infinite) sets $\mathbb{S}$ ordered by the relation $\preccurlyeq_{\oplus}$ (which is extended to $\mathbb{S}^N$), any asynchronous iteration converges for any initial guess $\mathtt{X}_0 \in \mathbb{S}^N$ if the following conditions are verified (see Proposition 3, p. 599 in [30] concerning finite or infinite sets): (a) $\mathtt{F}$ is closed on $\mathbb{S}^N$; (b) the synchronous iterations converge, and $\mathtt{X}_{n+1} \preccurlyeq \mathtt{X}_n$ for each $n \in \mathbb{N}$; (c) $\mathtt{F}$ is monotonous on $\mathbb{S}^N$, that is, for all $\mathtt{X}$ and $\mathtt{Y}$ in $\mathbb{S}^N$, $\mathtt{X} \preccurlyeq_{\oplus} \mathtt{Y}$ implies $\mathtt{F}(\mathtt{X}) \preccurlyeq_{\oplus} \mathtt{F}(\mathtt{Y})$.

When these conditions are fulfilled, $\mathsf{F}$ has a fixed point in $\mathbb{S}^N$ such that every asynchronous iteration corresponding to Eq. (4) converges.

In order to prove that our algorithm is self-stabilizing, we need to prove that for any underlying topology, for any initial vector configuration (the initial values in the RES registers), and for any successive choices of the fully distributed demon, its convergence is assured. Note that Eq. (4) is more general than Eq. (3) ($D_n[i]=n-1$ for any index $i$ and any step $n$). In turn, Eq. (3) is more general than Eq. (2) ($\mathscr{J}_n=\{1,\ldots,N\}$ for any step $n$). This order on equations mimics the total order on demons we mentioned in Section 2.3. Consequently, the proof of stabilization given in the following section is only established for the fully distributed demon.

## 4. Proving self-stabilization using path algebra

In this section we prove the self-stabilization property of the parametric algorithm instantiated by a strictly idempotent $r$-operator, using path algebra and asynchronous iterations, under the fully distributed demon.

**Theorem 9.** *When the parametric algorithm $\mathscr{PA}$ is instantiated with a strictly idempotent $n$-ary $r$-operator, it is self-stabilizing on any topology.*

**Proof.** An $n$-ary $r$-operator $\triangleleft$ defined on the set $\mathbb{S}$ is built from an $s$-operator $\oplus$ and $n-1$ $r$-mappings $\mathsf{r}_i$ which are endomorphisms of $\mathbb{S}$ (see Hypothesis 6):

$$\triangleleft(x_0,\ldots,x_{n-1}) = x_0 \oplus \mathsf{r}_1(x_1) \oplus \cdots \oplus \mathsf{r}_{n-1}(x_{n-1}).$$

Such an operator leads to the following local computations on nodes of the distributed system (see Section 3.2.4):

$$\mathsf{RES}[v] = \mathsf{ROM}[v] \oplus \mathsf{r}_1(\mathsf{RES}[u_1]) \oplus \cdots \oplus \mathsf{r}_{\delta^-(v)}(\mathsf{RES}[u_{\delta^-(v)}]).$$

Let $(\mathbb{H},\oplus,\otimes)$ be the idempotent semiring of the endomorphisms over the semigroup $(\mathbb{S},\oplus)$, and $(\mathbb{M},\oplus,\otimes,\cdot)$ be the semi-algebra of the $N\times N$ matrices composed of elements of $\mathbb{H}$ (see Section 3.3). Let $\mathbf{A}\in\mathbb{M}$ be the $N\times N$ precedence matrix associated to the distributed system, where each entry $\mathbf{A}[i][j]$ is the $r$-mapping corresponding to the edge $(j,i)$ if it exists, and $\mathsf{e}_\oplus$ else (see Sections 3.3 and 3.4).

Assuming the fully distributed demon, and using asynchronous iterations (see Section 3.4.4), the global computations performed by the whole distributed system can be modeled as follows:

$$\mathtt{X}_{n+1}[i] = \begin{cases} \mathtt{X}_n[i] & \text{if} \quad i \notin \mathscr{J}_n, \\ (\mathsf{F}((\mathtt{X}_{D_n[1]}[1],\ldots,\mathtt{X}_{D_n[N]}[N])^{\mathrm{t}}))[i] & \\ \quad = (\mathbf{A}\otimes(\mathtt{X}_{D_n[1]}[1],\ldots,\mathtt{X}_{D_n[N]}[N])^{\mathrm{t}} \oplus \mathtt{B})[i] & \text{if} \quad i \in \mathscr{J}_n. \end{cases} \tag{5}$$

Proving the self-stabilization of Algorithm 1 on any topology is equivalent to proving the convergence of these asynchronous iterations with delay for *any initial vector* $\mathtt{V}_0$ and any precedence matrix $\mathbf{A}$. The vector operator $\mathtt{F}$ is defined on $\mathbb{S}^N$, which is a Cartesian product. We then have to verify that the Üresin and Dubois following conditions hold (see Section 3.4.5):

(1) the vector operator $\mathtt{F}$ is closed on $\mathbb{S}^N$;
(2) $\mathtt{F}$ is monotonous on $\mathbb{S}^N$;
(3) the synchronous iterations converge.

We successively prove that these conditions are verified.

1. From definition of $\mathtt{F}$, the first condition is straightforward.
2. Let $\mathtt{X}_1$ and $\mathtt{X}_2$ be two vectors of $\mathbb{S}^N$ such that $\mathtt{X}_1 \preccurlyeq_\oplus \mathtt{X}_2$. We then have $\mathtt{X}_1 \oplus \mathtt{X}_2 = \mathtt{X}_1$ and $\mathtt{X}_1 \otimes \mathbf{A} \oplus \mathtt{X}_2 \otimes \mathbf{A} = \mathtt{X}_1 \otimes \mathbf{A}$ and $(\mathtt{X}_1 \otimes \mathbf{A} \oplus \mathtt{B}) \oplus (\mathtt{X}_2 \otimes \mathbf{A} \oplus \mathtt{B}) = \mathtt{X}_1 \otimes \mathbf{A} \oplus \mathtt{B}$. The second condition is thus fulfilled.
3. There remains to prove that synchronous iterations $\mathtt{X}_{k+1} = \mathbf{A} \otimes \mathtt{X}_k \oplus \mathtt{B}$ converge for *any initial guess* $\mathtt{X}_0$.

We have

$$\begin{aligned}
\mathtt{X}_1 &= \mathbf{A} \otimes \mathtt{X}_0 \oplus \mathtt{B},\\
\mathtt{X}_2 &= \mathbf{A} \otimes \mathtt{X}_1 \oplus \mathtt{B}, = \mathbf{A}^2 \otimes \mathtt{X}_0 \oplus \mathbf{A} \otimes B \oplus \mathtt{B},\\
&\vdots
\end{aligned}$$

which leads to

$$\begin{aligned}
\mathtt{X}_k &= \mathbf{A}^k \otimes \mathtt{X}_0 \oplus \mathbf{A}^{k-1} \otimes \mathtt{B} \oplus \cdots \oplus \mathbf{A} \otimes \mathtt{B} \oplus \mathtt{B}\\
&= \mathbf{A}^k \otimes \mathtt{X}_0 \oplus (\mathbf{A}^{k-1} \oplus \cdots \oplus \mathbf{A} \oplus \mathbf{E}_\otimes) \otimes \mathtt{B}\\
&= \mathbf{A}^k \otimes \mathtt{X}_0 \oplus \mathbf{A}^{(k-1)} \otimes \mathtt{B} \quad \text{(see Section 3.3).} \qquad (6)
\end{aligned}$$

Let $\mathsf{r}_\perp$ be the largest element of $\mathbb{H}$ which is smaller than all the component of the matrix $\mathbb{A}$ (in sense of $\preccurlyeq_\oplus$): $\mathsf{r}_\perp \stackrel{\text{def}}{=} \bigoplus_{1 \leqslant i,j \leqslant n} \mathbf{A}[i][j]$. Let $\mathbf{R}_\perp$ be the matrix of $\mathbb{M}$ defined by $\mathbf{R}_\perp[i][j] = \mathsf{r}_\perp$, for all indices $i$ and $j$. We have: $\mathbf{R}_\perp \preccurlyeq_\oplus \mathbf{A}$ which leads to $\mathbf{R}_\perp^k \preccurlyeq_\oplus \mathbf{A}^k$. Thanks to hypothesis 8, we have $\lim_{k\to+\infty} \mathsf{r}_\perp^k = \mathsf{e}_\oplus$. Hence $\lim_{k\to+\infty} \mathbf{R}_\perp^k = \mathbf{E}_\oplus$. Thus $\lim_{k\to+\infty} \mathbf{A}^k = \mathbf{E}_\oplus$.

Since the $r$-operator is strictly idempotent (Hypothesis 7), $\mathsf{h}_\otimes \oplus \mathsf{r}_i = \mathsf{h}_\otimes$ for all $i \in \{1, \ldots, |E|\}$ and there is no absorbing circuit in the network. Thus $\lim_{k\to+\infty} \mathbf{A}^{(k-1)} = \mathbf{A}^*$ (see Section 3.3). We then have from Eq. (6)

$$\begin{aligned}
&\forall \mathtt{X}_0 \in \mathbb{S}^n, \quad \exists k_0 \in \mathbb{N} \quad \text{such that} \quad \text{for } k \geqslant k_0,\\
&\mathtt{X}_k = \mathbf{A}^k \otimes \mathtt{X}_0 \oplus \mathbf{A}^{(k-1)} \otimes \mathtt{B} = \mathbf{A}^* \otimes \mathtt{B}
\end{aligned}$$

and the synchronous iteration $\mathtt{X}_{k+1} = \mathbf{A} \otimes \mathtt{X}_k \oplus \mathtt{B}$ converges for *any initial guess* $\mathtt{X}_0$. The third condition is then verified.

From results of [30], the theorem is proved. □

## 5. Applications

In this section, we briefly give some examples of $r$-operators designed to solve particular problems. In [19], several applications that use $r$-operators based on $s$-operators that define total order relations are given. These results remain valid here because we have a weaker requirement on the $s$-operator (defining a partial ordering relation is sufficient). Among the operators presented in [19], **minc** is defined as $\mathbf{minc}(x, y) = \mathbf{min}(x, y + 1)$, and is used to solve distance computation, shortest dipath spanning tree and forest. Others problems such as single and multiple source shortest paths, depth-first-search tree are solved using operators $\mathbf{minc}_w$ and **lexicat**, respectively. In the following, we give other examples of $r$-operators that solve different problems: the best reliable path from some transmitters is a variant of the shortest paths problem that uses a different "metric", while the ordered ancestor list is an application that was not possible in the framework of [19] (since the $s$-operator it is based on induces only a partial ordering relation).

### 5.1. Best reliable path from some transmitters

Assume that $\tau_v^i$ is the failure rate on the $i^h$ incoming edge of the node $v$ and $0 < \tau_v^i \leqslant 1$. We then denote by $\overline{\tau_v^i}$ the reliable rate of this edge: $\overline{\tau_v^i} = 1 - \tau_v^i$. The reliable rate of a path is the product of the reliable rate of all its edges. We define the $n$-ary $r$-operator $\mathsf{Maxtimes}_{\bar{\tau}}$ on $\mathbb{S} = [0,1] \cap \mathbb{R}$ by $(n = \delta^{-(v)} + 1)$

$$\mathsf{Maxtimes}_{\bar{\tau}}(x_0, \ldots, x_{\delta^-(v)}) = \mathsf{Max}(x_0, x_1 \times \overline{\tau_v^1}, \ldots, x_{\delta^-(v)} \times \overline{\tau_v^{\delta^-(v)}}).$$

The $n$-ary $r$-operator $\mathsf{Maxtimes}_{\bar{\tau}}$ is based on the $s$-operator $\mathsf{Max}$ that defines on $\mathbb{S}$ a total ordering relation $\preccurlyeq_{\mathsf{Max}}$ which is, in fact, the usual order $\geqslant$. Moreover, each $r$-mapping $r_v^i(x) = x \times \overline{\tau_v^i}$ verifies $x \prec_{\mathsf{Max}} r_v^i(x)$ (which means that $x > r(x)$). Thus the $n$-ary $r$-operator is strictly idempotent. In addition, Hypothesis 8 holds. When $\mathsf{ROM}[v] = 1$ if $v$ is a transmitter and $\mathsf{ROM}[v] = 0$ else, the best reliable path problem is solved after stabilization of Algorithm 1. Indeed, the best path is maintained by the knowledge, on each node, of one incoming variable containing the smallest datum of all those in the incoming variables.

### 5.2. Ordered ancestor list

We provide in this section an operator that maintains an ordered list of ancestors for any node $v$ in the network. Assuming that nodes have unique indices over the network, let $\mathbb{S}$ be the set of lists of sets of indices. For example, if $a$, $b$ and $c$ are processor indices, then the list $(\{b\}, \{a, c\})$ is an element of $\mathbb{S}$. If a node $v$ in

the network $G$ obtains this element, it should be interpreted as follows: $d_G(b,v)=1$, $d_G(a,v)=d_G(c,v)=2$ and $v$ has no more ancestors in $G$. The aim of an ordered ancestor list algorithm is to build such a list for all the nodes of the network. During the execution, all lists received by a node should be merged with its current list after they have been shifted by one. We will now exhibit an $r$-operator allowing to design a self-stabilizing protocol for this problem.

In order to only consider potentially useful lists, we define an equivalence relation $\equiv$ such that there is no repeated term nor empty set in lists of $\mathbb{S}$: $(S_1,\ldots,S_k)\equiv(S_1,S_2\backslash S_1,\ldots,S_k\backslash(S_1\cup\cdots\cup S_{k-1}))$ and $(S_1,\ldots,S_k,\emptyset,S_l,\ldots,S_m)\equiv(S_1,\ldots,S_k)$ for $k\geqslant 1$ (note that lists beginning by an emptyset are allowed). Next we consider the $s$-operator $\oplus$ on $\mathbb{S}/_{\equiv}$ which merges term to term elements of lists of $\mathbb{S}/_{\equiv}$. For example: $(\{d\},\{b\},\{a,c\})\oplus(\{c\},\{a,e\},\{b\})=(\{d,c\},\{b,a,e\},\{a,c,b\})\equiv(\{d,c\},\{b,a,e\})$. Moreover, we consider the endomorphism $\mathsf{r}$ from $\mathbb{S}/_{\equiv}$ to $\mathbb{S}/_{\equiv}$ which maps list $l=(S_1,\ldots,S_k)$ to list $\mathsf{r}(l)=(\emptyset,S_1,\ldots,S_k)$. For example: $\mathsf{r}((\{d\},\{b\},\{a,c\}))=(\emptyset,\{d\},\{b\},\{a,c\})$.

Let $\mathsf{Ant}$ be the binary $r$-operator defined with the $s$-operator $\oplus$ and the $\mathsf{r}$-mapping as $\mathsf{Ant}(l_1,l_2)=l_1\oplus\mathsf{r}(l_2)$. Operator $\mathsf{Ant}$ is strictly idempotent because (i) $(S_1,\ldots,S_k)\oplus(\emptyset,S_1,\ldots,S_k)=(S_1,\ldots,S_k)$ and (ii) $(S_1,\ldots,S_k)\neq(\emptyset,S_1,\ldots,S_k)$ leads to (iii) $(S_1,\ldots,S_k)\preccurlyeq_{\oplus}\mathsf{r}(S_1,\ldots,S_k)$. From our theorem and Hypothesis 8, we can conclude that any execution satisfies the specification. Indeed, when the $\mathsf{ROM}$ of each node $v$ contains list $(\{v\})$, the result in $\mathsf{RES}[v]$ after stabilization is $\mathsf{RES}[v]=\bigoplus\{r^{\mathrm{d}_G(u,v)}(\{u\}),u\in\Gamma_G^-(v)\}$. This expression is a complete representation of $v$'s ancestors ordered through distance to $v$.

## 6. Conclusion

Describing distributed systems using $r$-operators is convenient due to the local expression of computations occuring in the global system. When actually implementing scalable distributed algorithms, each processor local code has no knowledge of the global topology or configuration. In addition, in strongly connected networks, the ordered ancestor list construction presented in Section 5.2 builds at each node the list of all nodes in the network. Using the scheme presented in [13] on top of our algorithm, we are then able to solve *any* global computation task in a self-stabilizing way.

Max-plus algebra and the matrix representation take the orthogonal approach, using global entities to model the whole system configuration. This permits proving very fine conditions on both circuits and associated operators in order that the distributed system converges to a desirable configuration. While having a condition on each local operator (being a strictly idempotent $r$-operator) is stronger than having a condition on each circuit and associated operators (no absorbing circuit), it is easier using our approach (i) to write generic proofs that work on any topology directed graphs, and (ii) to verify that a given algorithm satisfies the conditions.

The main contribution of this paper is the connection between asynchronous iterations and their matrix representation for the one hand, distributed systems and their different scheduling policies for the other hand. This connection was done thanks to max-plus algebra, which permitted to enhance significantly previous results in distributed computing, such as [4,19].

## References


[1] Y. Afek, A. Bremler, Self-Stabilizing Unidirectional Network Algorithms by Power Supply, MIT Press, Cambridge, MA, 1998; Chicago J. Theoret. Comput. Sci. 3 (1998).

[2] Y. Afek, S. Kutten, M. Yung, Memory-efficient self-stabilization on general networks, Proc. WDAG'90, 1990, pp. 15–28.

[3] L.O. Alima, J. Beauquier, A.K. Datta, S. Tixeuil, Self-stabilization with global rooted synchronizers, Proc. ICDCS'98, 1998.

[4] A. Arora, P. Attie, M. Evangelist, M.G. Gouda, Convergence of iteration systems, Distributed Comput. 7 (1993) 43–53.

[5] B. Awerbuch, B. Patt-Shamir, G. Varghese, Self-stabilization by local checking and correction, Proc. FOCS'91, Los Alamitos, CA, October 1991, pp. 268–277.

[6] B. Awerbuch, B. Patt-Shamir, G. Varghese, S. Dolev, Self-stabilization by local checking and global reset, in: Internat. Workshop on Distributed Algorithms, Berlin, 1994, pp. 326–339.

[7] F. Baccelli, G. Cohen, G. Olsder, J.-P. Quadrat, Synchronisation and Linearity, an Algebra for Discrete Event Systems, Series in Probability and Mathematical Statistics, Wiley, Chichester, UK, 1992.

[8] G.M. Baudet, Asynchronous iterative methods for multiprocessors, J. ACM 25 (2) (1978) 226–244.

[9] A. Costello, G. Varghese, Self-stabilization by window washing, Proc. PODC'96, 1996, pp. 35–44.

[10] S.K. Das, A.K. Datta, S. Tixeuil, Self-stabilizing algorithms on DAG structured networks, Parallel Processing Letters 9 (1999) 563–574.

[11] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644.

[12] S. Dolev, G. Gouda, M. Schneider, Memory requirements for silent stabilization, Proc. PODC'96, 1996, pp. 27–34.

[13] S. Dolev, T. Herman, Superstabilizing Protocols for Dynamic Distributed Systems, MIT Press, Cambridge, MA, 1997; Chicago J. Theoret. Comput. Sci. 4 (1997).

[14] S. Dolev, A. Israeli, S. Moran, Self stabilization of dynamic systems assuming only read/write atomicity, Distributed Comput. 7 (1993) 3–16.

[15] S. Dolev, A. Israeli, S. Moran, Resource bounds for self-stabilizing message-driven protocols, SIAM J. Comput. 26 (1997) 273–290.

[16] S. Dolev, D.K. Pradhan, J.L. Welch, Modified tree structure for location management in mobile environments, Proc. INFOCOM'95, vol. 2, 1995, pp. 530–537.

[17] B. Ducourthial, New operators for computing with associative nets, Proc. SIROCCO'98, Amalfi, Italia, 1998.

[18] B. Ducourthial, Les Réseaux Associatifs (Associative nets), Ph.D. Thesis, Université de Paris Sud, France, January 1999.

[19] B. Ducourthial, S. Tixeuil, Self-stabilization with *r*-operators, Distributed Computing. 14 (2001) 147–162.

[20] M. Gondran, Algèbre linéaire et cheminement dans un graphe, RAIRO, Inform. Théorique 9 (1) (1975) 77–99.

[21] M. Gondran, M. Minoux, Graphes et Algorithmes, No. 37 in Collection de la DER de EDF, Eyrolles, 1979, Also published as Graphs and Algorithms, Wiley, New York, 1986.

[22] A. Israeli, M. Jalfon, Uniform Self-Stabilizing Ring Orientation, Information and Computation, vol. 104, Academic Press, New York, 1993, pp. 175–196.

[23] M. Jayaram, G. Varghese, Crash failures can drive protocols to arbitrary states, Proc. PODC'96, 1996, pp. 247–256.

[24] M. Minoux, Structure algébrique généralisée des problèmes de cheminement dans les graphes: théorèmes, algorithmes et applications, Rech. Opér. RAIRO 10 (2) (1976).

[25] M. Schneider, Self-stabilization, ACM Computing Surveys 25 (1993) 45–67.

[26] G. Tel, Topics in Distributed Algorithms, Cambridge International Series on Parallel Computation, vol. 1, Cambridge University Press, Cambridge, 1991.

[27] G. Tel, Introduction to Distributed Algorithms, Cambridge University Press, Cambridge, 1994.

[28] M.-S. Tsai, S.-T. Huang, A self-stabilizing algorithm for the shortest paths problem with a fully distributed demon, Parallel Process. Lett. 4 (1& 2) (1994) 65–72.

[29] A. Üresin, M. Dubois, Sufficient conditions for the convergence of asynchronous iterations, Parallel Comput. 10 (1989) 83–92.
[30] A. Üresin, M. Dubois, Parallel asynchronous algorithms for discrete data, J. ACM 37 (3) (1990) 588–606.
[31] S. Dolev, Self-Stabilization, MIT Press, Cambridge, MA, 2000.